Protected Software Module Architectures

Raoul Strackx[1] · Job Noorman[1] · Ingrid Verbauwhede[2] · Bart Preneel[2] · Frank Piessens[1]

[1]iMinds-DistriNet, KU Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
{Raoul.Strackx | Job.Noorman | Frank.Piessens}@cs.kuleuven.be

[2]iMinds-COSIC, KU Leuven
Kasteelpark Arenberg 10, 3001 Leuven, Belgium
{Ingrid.Verbauwhede | Bart.Preneel}@esat.kuleuven.be

Abstract

A significant fraction of Internet-connected computing devices is infected with malware. With the increased connectivity and software extensibility of embedded and industrial devices, this threat is now also relevant for our industrial infrastructure and our personal environments. Since many of these devices interact with remote parties for security-critical or privacy sensitive transactions, it is important to develop security architectures that allow a stakeholder to assess the trustworthiness of a computing device, and that allow such stakeholders to securely execute software on that device. Over the past decade, the security research community has proposed and evaluated such architectures. Important and promising examples are *protected software module architectures.* These architectures support the secure execution of small protected software modules even on devices that are malware infected. They also make it possible for remote parties to collect *trust evidence* about a device; the remote party can use the security architecture to collect measurements that give assurance that the device is in a trustworthy state.
In this paper we outline the essential ideas behind this promising recent line of security research, and report on our experiences in developing several protected module architectures for different types of devices.

1 Introduction

Any programmable device is at risk of being reprogrammed, exploited, or infected with malware by attackers. This risk goes up significantly for network-connected devices, as exploitation or infection can now be done remotely. Protection against this threat is significantly harder for devices that are *open* in the sense that they support software extensibility, possibly even by several parties that do not necessarily trust each other. Historically, the first such devices were classical computers (desktops and servers), and history has taught us that the threat of malware infection and other software attacks against these devices is very real indeed.

Over the past years, more and more embedded computing devices are being connected to the Internet, and many of these devices are open to some extent to software extensibility. Examples include smartcards that support over-the-air updates, programmable sensor-networks, set-top boxes and internet-connected TV's as well as SCADA (supervisory control and data acquisition) systems that control important components of our critical infrastructure. The increasing connectivity of all these devices, as well as their increasing impact on our society,

gives rise to significant threats. Viega and Thompson [ViTo12] describe several recent incidents and summarize the state of embedded device security as “a mess”.

An important research question in this context is: what kind of infrastructural support for security (such as hardware support, or operating system support for security) will make it easier for device manufacturers to construct secure networked and extensible devices? Of course, we have built up a rich body of experience on securing classical computing devices, such as servers or desktops. A first important technique is the use of hardware support for virtual memory, and processor privilege levels. An operating system can build on this support to run software modules in isolated processes, and the operating system can guard the interactions between these various processes. A second important technique is the use of a memory-safe virtual machine (for instance a Java VM) where software modules are deployed in memory-safe bytecode, and a security architecture in the VM guards the interactions between them. While these well-understood techniques could be ported to the new context of networked embedded devices, there are also several important known disadvantages to these techniques. First, there is the cost in terms of required resources such as chip surface, power or performance that might be unacceptable in some embedded device scenarios. Second, these classic solutions all require the presence of a sizable trusted software layer (either the operating system, or the VM implementation). History has shown that it is very difficult to get such a software layer sufficiently secure that it cannot be exploited by software attacks such as buffer overflows or injection attacks [YJP12]. And finally, with these classic solutions it is non-trivial to securely support *remote attestation*, where a stakeholder can remotely check that a specific software module is running untampered on a remote device.

As a consequence, researchers have started investigating alternatives. One important line of research is developing *protected (software) module architectures* [MPP+08, MLQ+10, StPi12]. These are security architectures running independent of a classical operating system, and that can execute security sensitive code in an isolated area of the system. The isolation does not rely on the operating system, thus improving the security guarantees that can be offered. Of course, an important design goal (and design challenge) is to realize this while remaining compatible with current operating systems and hardware. Most of these proposed systems leverage recent hardware extensions for trusted computing or virtualization to execute code. These architectures were originally developed for high-end computing devices such as desktops and servers, to better protect against the malware threat.

A second important line of research is the development of program-counter based memory access control systems for isolation. This is an alternative kind of memory protection that is less expensive than full support for virtual memory, yet it is sufficient to implement strong isolation between software modules, and it is very compatible with remote attestation. Several researchers have independently proposed program-counter based memory access control as a suitable memory protection mechanism for low-end embedded devices [EFPT12, StBP10].

In this paper, we report on our experiences in developing several protected software module architectures that build on the idea of program-counter based memory access control. We show that the combined idea of protected modules using program-counter based access control can be useful for a range of systems, ranging from low-end embedded devices (where memory protection would be built into the hardware) over desktops and servers (where memory protection can be either realized by means of a hypervisor or in an operating system kernel). In general, these new security architectures contribute to increasing the trust that stakeholders can have in a networked computing device, either by providing strong assurance about the state of a device (as in remote attestation), or by supporting measurements of the state of the device that can be used as heuristic evidence of trustworthiness.

2 Program-Counter based Memory Access Control

Program-counter based memory access control is a memory protection technique intended to provide isolation between software modules running on the same device, but that do not necessarily trust each other. As such, it is a low-cost alternative to virtual memory, processor privilege levels and process isolation. We first explain how program-counter based memory access control works, and then we show that it provides a very strong and precise notion of isolation between modules.

2.1 Software Modules and Memory Access Control

Software modules are essentially memory sections. One module consists of two sections: a text (or code) section containing protected code and constants and a protected data section. In addition, the text section has a fixed number of *entry points*. These are addresses of memory locations in the text section where control flow is allowed to enter the module.

from \ to	*Protected*			*Unprotected*
	Entry point	*Code*	*Data*	
Protected	r x	r x	r w	r w x
Unprotected	x			r w x

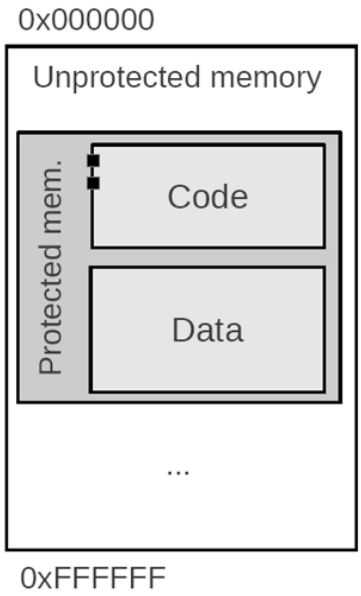

Fig. 1: A software module and the memory access control rules

Figure 1 shows a memory space with one protected module with two entry points in the code section. The Figure also summarizes the memory access control rules that program-counter based memory access control enforces. While some details of this access control matrix vary in different implementations, the key characteristic is that access rights to memory, depend on where the processor is executing (i.e. on the value of the program counter). If the program counter is in unprotected memory then only unprotected memory can be read or written. Execution can continue to other unprotected memory, or to one of the entry points of a protected module. After jumping to an entry point, the program counter is now in the code section of a protected module, and the protected data section of that module can be read or written, and execution can proceed to any address of the protected code section of that module. In addition, any access to unprotected memory is allowed.

When there is more than one protected module in memory, the rules for accessing the code and data sections of a given module treat all the other modules as if they were unprotected memory. So protected module A can only jump to the entry points of module B; it cannot read or write the data section of module B, or start executing in the code section at any other address than an entry point.

The key underlying idea of this memory access control model is that it ensures that *only the code of a given module can manage the state of that specific module*.

Isolation Properties

Program-counter based memory access control is a sufficiently strong building block to get all the isolation guarantees that modern programming languages can express at source code level [ASJP12]. Modern high-level programming languages such as Java, C#, ML or Haskell offer protection facilities such as abstract data types, the private field modifier, or module systems. These programming language concepts were designed to enforce software engineering principles such as information hiding and encapsulation. But these can also be used as building blocks to ensure security properties of programs. For instance, declaring a class instance variable private in Java protects the integrity and confidentiality of that field towards instances of other classes.

Unfortunately, these protection features are typically lost when the program is compiled. Suppose for instance that we compile a Java program to native machine code. Then, an attacker that has injected malware in the memory space of the program can read or write any private variable, thus violating that variable's confidentiality and integrity. In other words, the isolation guarantees offered by the source language can be violated.

However, recent research [ASJP12] has shown that it is possible to maintain the security properties of a high-level program even after it is compiled into a lower-level language (such as native code), by relying only on program-counter based memory access control. This is strong evidence that this kind of memory access control is sufficient for all practical purposes. We can get the isolation properties that modern high-level programming languages support.

3 Three Implementations

The idea of program-counter based memory access control can be implemented in different ways. In this Section, we discuss our experience with three implementations, each of these with its own advantages and limitations:

- *Sancus* is a hardware-level implementation, that in addition implements a very strong form of remote attestation
- *Fides* is a hypervisor-level implementation, that shows that the additional isolation offered by program-counter based memory access control can also be efficiently implemented on legacy desktops and servers
- *Salus* is a kernel-level implementation that offers program-counter based memory access control as an additional layer of isolation on top of regular virtual memory

We briefly discuss each of the three implementations.

3.1 Sancus

The Sancus architecture [NAD+13] implements program-counter based access control in hardware in a microprocessor, and in addition implements a remote attestation mechanism.

The architecture targets systems where a single infrastructure provider, IP, owns and administers a (potentially large) set of microprocessor-based systems that we refer to as nodes N_i. A variety of third-party software providers SP_j are interested in using the infrastructure provided by IP. They do so by deploying software modules $SM_{j;k}$ on the nodes administered by IP.

A Sancus node N (Fig. 2) is a low-cost, low-power microcontroller (our implementation is based on the TI MSP430). The processor in the nodes uses a von Neumann architecture with a single address space for instructions and data. The processor is extended with a protected storage area that, for each protected software module loaded in main memory, maintains the following metadata:

- The bounds of the text and data sections of the module. The processor uses these to enforce the memory access control rules.
- A cryptographic module key $K_{N,SP,SM}$ that is used for remote attestation and authenticated communication.

The hardware ensures that the module key $K_{N,SP,SM}$ can only be used by (1) a software module SM running on a specific node N on behalf of the software provider SP, and (2) by the SP itself. It does so by deriving the key from the hash of the contents of the text section of the module (among other things), thus guaranteeing that any tampering with the module would invalidate the key. This is the basis for remote attestation support and authenticated communication in Sancus.

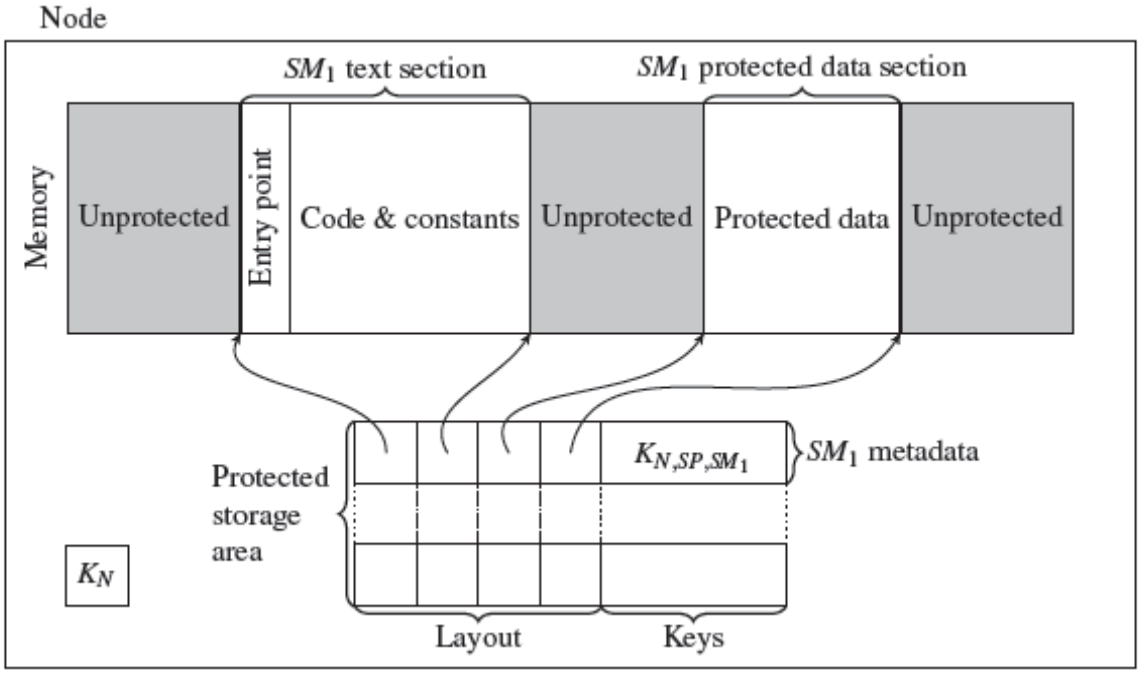

Fig. 2: A Sancus node (from [NAD+13])

The essence of the security argument for remote attestation in Sancus is the following: the module key $K_{N,SP,SM}$ can only be used by means of specific processor instructions, and the hardware of node N ensures that the use of key $K_{N,SP,SM}$ is limited to code from module SM. In other words, some form of program-counter based access control to cryptographic operations is enforced to ensure that only the correct module can use a module key. For a detailed description of the system, and a security and performance evaluation, we refer the reader to [NAD+13].

3.2 Fides

The Fides architecture [StPi12] implements program-counter based access control in a hypervisor running on top of a commodity desktop processor. It shows that the idea of program-counter based access control can be used efficiently on modern processors, even if one cannot change the hardware.

Figure 3 sketches the implementation strategy: A hypervisor runs two separate virtual machines that both have the same logical view of physical memory, but with different memory access control rights. The Legacy virtual machine (on the right) runs the legacy operating system and legacy software. The memory belonging to protected software modules (called Self-Protecting Modules or SPM's in Fides) is made inaccessible in this legacy virtual machine, and any attempt to access it will trap to the hypervisor. The hypervisor then verifies the memory access control rules (in this case, it checks if an appropriate entry point was called) and if so switches to the secure virtual machine (on the left). That second virtual machine enables access to the memory regions belonging to the module that was just entered. A small dedicated kernel in the second virtual machine enforces the memory access control rules between modules.

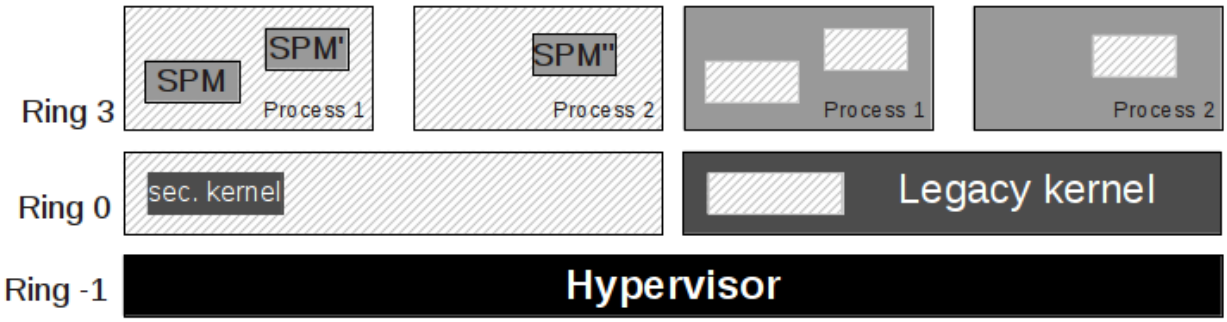

Fig. 3: The implementation strategy behind Fides (from [StPi12])

For a detailed description of the Fides system, and a security and performance evaluation, we refer the reader to [StPi12].

3.3 Salus

Our third and final implementation is Salus, a kernel-based implementation. Salus applies the idea of program-counter based memory access control in the virtual address space of processes running on a Linux operating system. In other words, the isolation offered by program-counter based memory access control is used as *an additional layer of protection* over the isolation offered by the process abstraction in the Linux operating system.

In Salus, the effects of program-counter based access control are obtained by making sure that a trap to the operating system kernel occurs on entering and exiting of a module. On entering, this is ensured by setting memory protection of modules to no-access as long as the module is not active. Hence, attempting to enter will trap with a memory access violation to the kernel that can then check if an entry point is being called and if so change the memory access rights. On exit of a module this is achieved by means of a new system call that resets memory access rights before proceeding with execution outside of the module.

Similar to how Sancus extends the idea of program-counter based access control to cryptographic operations offered by the hardware, Salus extends this to system calls. Protected

modules can be configured with a policy that says which system calls are allowed to occur within the module. This opens the possibility to use Salus not only for protecting a module from its environment, but also for enforcing modules to operate under a least-privilege regime.

For a detailed description of the Salus system, we refer the reader to the master thesis of Niels Avonds [Avon13].

4 Application scenarios

Program-counter based access control is a generic software isolation mechanism, and it can be used in a variety of application scenarios. The three different implementations we have discussed have different advantages and limitations, and depending on the application scenario, one implementation might be preferable over the other. In this Section, we list a few possible application scenarios, and discuss what implementation technique fits best with them.

4.1 Trustworthy extensible networked systems

All kinds of computing devices play a crucial role in our society and in our individual lives, both private and professional. On the high-end side, there is the emergence of cloud computing, while on the low-end side there is an explosion of shrinking or disappearing devices: a modern car has over 100 micro-controllers, RFID chips are replacing barcodes, and sensor networks are common in logistics, healthcare and many other application domains.

Many of these networked computing devices are software extensible by a variety of stakeholders. Any system that supports extensibility (through installation of software) by several stakeholders must implement measures to make sure that the different software modules cannot interfere with each other in undesired ways (either because of bugs in the software, or because of malice).

We believe such extensible networked systems are the prime application scenario for the protected module architectures described in this paper. Since both minimization of the trusted software stack, as well as support for remote attestation of the state of a device are important requirements for this application scenario, a hardware-level, Sancus-style implementation seems the most appropriate. In particular for small embedded devices, where hardware heterogeneity is common, customization of the hardware to support Sancus could be a feasible path.

4.2 Isolating security critical components of applications

A more short-term application scenario considers the protection of desktops and servers against the malware threat. Given the prevalence of malware-infections on the Internet, it makes sense to deploy additional protection for security critical components (like cryptographic libraries) running on these systems, to make sure that secret keys cannot be stolen from such a component even if the system is infected.

This scenario was the original motivating scenario for protected module architectures such as Flicker [MPP+08] and TrustVisor [MLQ+10]. By implementing program-counter based memory access control in a hypervisor, as in Fides, one can get such an additional protection with the additional benefit that – because of the shared memory address space between protected modules and the rest of the program – porting legacy programs to such a protected module architecture is made simpler.

4.3 Trust assessment modules

For existing legacy systems, it may be infeasible to redesign and re-implement them with better isolation and/or attestation support. Changing the hardware, or even refactoring the application code to support protected modules, might be too costly. Yet, even for such systems, it is important for remote stakeholders to gather evidence about the trustworthiness of a remote networked system.

In a third application scenario, the protected software module architecture is used to protect modules that are added to the system purely for the purpose of gathering trust evidence. Since these security architectures are designed to give protected modules access to the entire unprotected legacy system state, they can be used to securely execute measurement modules that inspect the state of legacy software while being protected in case that legacy software has been infected by an attacker. In other words, modules have the necessary access to perform measurements, and are protected from tampering by the environment they are measuring.

Such *trust assessment modules* will analyze the state of other software applications and services on the system that they are part of to compute trust evidence, i.e. metrics that give an indication of the likelihood that a software application or service is in a trustworthy state and has not been compromised. They perform a function similar to intrusion detection or virus detection systems.

Since trust assessment modules run on top of a legacy system, it is mainly the Fides and Salus implementations that are useful for this application scenario. Fides is to be preferred, as it has a smaller trusted computing base, but Salus might be applicable and give better performance for cases where the system to be monitored is a single process (e.g. a server process). In that case, the trust assessment module can be made application-specific: it could monitor for instance whether state invariants that are known to be valid in the application are violated.

4.4 Containing application vulnerabilities

Software applications are often built from modules at source code level, but after compilation that modularization is lost. As a final application scenario, we consider the case where compilation maps source-level modules on protected modules. While this kind of compilation is difficult for classic isolation mechanisms (consider for instance the effort involved in refactoring a server program such that its modules run as different processes), this is easier for program-counter based memory access control, as the application still runs in one single address space.

An important advantage of this secure compilation is that vulnerabilities in a single module can now only impact that module. Hence, if we combine this with support for limiting the privileges of individual modules (as for instance in Salus, where the operating system calls available to modules can be limited), then we get an important additional layer of protection. By limiting the privileges of likely vulnerable modules (such as a parsing module, or a network-facing module), the bar is raised significantly for attackers, as exploiting the likely vulnerable module will give the attacker less privileges on the system under attack.

Both for performance reasons, as well as because of its potential to support least-privilege policies, this application scenario is best supported by kernel-level implementations such as Salus.

5 Limitations

While we hope that this paper convinces the reader of the potential value of protected software module architectures based on program-counter based memory access control, no security architecture is perfect, and the architectures described in this paper have limitations that require further research.

An important limitation is that the protection of a module depends on the correct implementation of that module: if a module contains a vulnerability, then protection guarantees can be undermined. For instance, if a module that implements a cryptographic library has a method that returns the key, then running that library inside a protected module offers no additional protection for key secrecy. Moreover, vulnerabilities in protected modules can be subtle and hard to detect. Hence, an important avenue for future work is the development of analysis and verification techniques for protected modules.

A second limitation is that it is difficult to marry protected modules and multi-threading in a secure way, in particular in cases where multiple modules collaborate. Multi-threading can give rise to time-of-check-to-time-of-use vulnerabilities, where one module checks the presence of another module before calling it, but between the check and the actual call another thread intervenes and removes the module that is called. Further research is needed to understand the interactions between multi-threading and program-counter based memory access control.

Finally, some important security concerns are not addressed yet by the protected module architectures described in this paper. In particular, the concern of *state-continuity* [PLD+11], i.e. the guarantee that an attacker cannot roll back a module to an earlier state by crashing and rebooting the system is an important avenue for future work.

6 Conclusion

With the combined trends of more network connectivity and more software extensibility for computing devices comes an increased threat of exploitation and malware infection. In this paper, we have surveyed a recent line of research that is developing countermeasures for this threat. Protected software module architectures provide additional protection against exploitation and can support the secure collection of trust evidence about the state of computing devices.

We have discussed three implementations of such protected software module architectures, and have considered a variety of application scenarios where these implementations might be useful. We also identified remaining limitations in these architectures that should be addressed by future research.

7 Acknowledgments

This work has been supported in part by the Intel Lab's University Research Office. This research is also partially funded by the Research Fund KU Leuven, and by the EU FP7 project NESSoS. With the financial support from the Prevention of and Fight against Crime Programme of the European Union (B-CCENTRE).

Raoul Strackx holds a PhD grant from the Agency for Innovation by Science and Technology in Flanders (IWT).

References

[ASJP12] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens: Secure compilation to modern processors, In: IEEE 25th Computer Security Foundations Symposium (CSF 2012), p. 171-185.

[Avon13] Niels Avonds: Implementation of a State-of-the-Art Security Architecture in the Linux Kernel. Master thesis KU Leuven, 2013.

[EFPT12] Karim El Defrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik: SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In Proceedings of the Network and Distributed System Security Symposium (NDSS 2012).

[MLQ+10] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig: TrustVisor: Efficient TCB Reduction and Attestation. In: IEEE Symposium on Security and Privacy 2010, p. 143-158.

[MPP+08] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki: Flicker: an execution infrastructure for tcb minimization. In: EuroSys 2008, p. 315-328.

[NAD+13] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens: Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base, In: 22nd USENIX Security symposium, 2013.

[PLD+11] Bryan Parno, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune: Memoir: Practical State Continuity for Protected Modules. In: Proceedings of the 2011 IEEE Symposium on Security and Privacy, p. 379-394.

[StBP10] Raoul Strackx, Frank Piessens, and Bart Preneel: Efficient isolation of trusted subsystems in embedded systems, In: SecureComm 2010, Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering: Security and Privacy in Communication Networks, volume 50, p. 1-18, 2010.

[StPi12] Raoul Strackx, Frank Piessens: Fides: Selectively hardening software application components against kernel-level or process-level malware, In: Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012), p. 2-13.

[ViTo12] John Viega, and Hugh Thompson: The state of embedded-device security (spoiler alert: It's bad). In: IEEE Security & Privacy Magazine, volume 10, issue 5, 2012, p. 68-70.

[YJP12] Yves Younan, Wouter Joosen, and Frank Piessens: Runtime countermeasures for code injection attacks against C and C++ programs, In: ACM Computing Surveys, volume 44, issue 3, p. 1-28, 2012.